Report Homework 3

Alex Pegoraro 227642
alex.pegoraro@studenti.unitn.it

December 14, 2023

Link to Git repository: `https://github.com/AlphaNightLight/parallel_homework3`

1 Parallel matrix multiplication

1.1 Introduction

The third homework deals with matrix multiplication in an environment with multiple machines. The request is to create a sequential program to multiply two matrices, and then parallelize it with MPI exploring different data partitions. In this report, I will focus on three possible solutions: divide the output matrix by rows, by columns and by blocks. The metrics I will deal with are strong and weak scaling, speedup, efficiency, and FLOPS.

1.2 Description and Methodology

1.2.1 The sequential program

The sequential program does not differ so much from its equivalent of the second homework. Again, the `main` function has just a wrapper role, as it instantiate the two random matrices throw auxiliary functions and then delegate the product task to the `matMul` function. This operation is done more times, to perform an average with the aim of reducing undesired spikes. Also the data structure to hold the matrix is the same of homework 2.

The execution time is measured with the help of the chrono library, and the value is stored in a report file in `.csv` format. The timestamps are sampled right before and right after the triple `for`, as allocations and prints shouldn't be included in the measure.

1.2.2 The parallel program with row partition

When we migrate to MPI, the first things to do are the inclusion of `mpi.h` library, the initialization (with also the retrieve of size and rank) and the finalization. Another important aspect, is to ensure that the data split we want to achieve is compatible with the thread number. As the first attempt is to divide A by rows, we should then check whether `ROW_N_A` is divisible by size, and if it's not the case we exit with an error message.

To be able to compute weak scaling, the code is wrapped in a `for` statement with two iterations, and if we are in the second one `ROW_N_A` is multiplied by the number of processors. Also note that to avoid a huge number of report files only the master writes the results.

In the innermost `for` we need to distinguish between the master and the other nodes. The master allocate the full A and B, and is also in charge of their initialization, as well as the storage of the time. The slaves, allocate the full matrix B, but only a part of A, namely the row stripe they will operate on. In both the cases the `matMulPar` function is invoked.

It's time to explain the data partition I'm trying to implement in this solution: the idea is to broadcast B over all the processes, while assigning to everyone only a bunch of rows of A. In such a situation, each node will have all the information it needs to compute the correspondent rows of C, invoking the `multiply` function over its local matrices, that simply implements the naive triple `for`.

In C++ dynamic allocation, all the rows of a matrix are stored in different and not necessarily contiguous memory locations. So, if we want to broadcast B we need to do it for each of its rows independently. Regarding A, the slaves should only wait to receive the various lines of their interest, but the master is slightly more complicated. It must iterate over all A, and for each line compute the correct rank to which it should be sent. When the master is rank zero, it will happen to have its target lines as the first lines of A, so it's enough to reduce the dimension of A to them[1]. In case the master rank is not zero, we also need to transfer its target lines at the beginning of A to be able to perform this trick.

Now, every processor has it local copy of A and B, and can obtain its part f C with the classical algorithm. To reconstruct C, we follow the opposite philosophy of A. The slaves, simply send their result to the master, line by line. The master needs first to enlarge C back to its full shape to be able to hold all the results, and if it is not rank 0 it moves the first lines of C (the ones it computed) into the position assigned to its rank. Then, it starts to wait for each line of C, expecting it from a particular source.

The time is measured with MPI's built in function `MPI_Wtime`. To have a fair idea of the time of our program, we can't measure the multiplication only, but also the message exchanges should be included in the analysis, as their overhead is absolutely not negligible. Hence, the samples occur before the first line of B is broadcast and after the final row of C is recieved.

1.2.3 The parallel program with column partition

If we have calculated C by rows, we can try to calculate it by columns. In this case, it will be A the matrix broadcasted, while B will be split into columns correspondingly with C. To distribute A, we simply broadcast all of its lines. On the other hand, to send B we can use a scatter this time[2], one for each row of B. For the same reason, a gather is enough to compose the final C.

Scatter and Gather require different buffers for send and receive, so each processor will have two support matrices: subB and subC. This means that slave processors will never use B and C, and so we can spare space not even allocating them. However, scatter requires a non-null sendbuf to work, and so B must be allocated. Furthermore, in parts of the algorithm we need to access B's rows and columns sizes. I solved this ambiguity allocating B as a 1×1 matrix, and then manually setting its dimension to the desired ones. Like this, B is non-null and properly dimensioned, but we don't waste space. in slave processors.

The logic for time measurement and mpi setup are the same of the previous partition, except for the fact that this time we should check `COL_N_B` to be compatible with the number of mpi

[1]Change the `A.rows` value does not affect the real dimension of A, it is simply bound the action of `multiply` to the specified number of lines.

[2]we couldn't use a scatter in the previous algorithm as we would have needed to extract the column vectors, that is not possible in the way dynamic matrices are stored in C++, i.e. row major.

processors involved, as we will split by this number. For the same reason, it is also the parameter that will be increased during weak scaling.

1.2.4 The parallel program with block partition

The homework explicitly suggests us to perform a data partition in which C is decomposed by blocks. To compute such a block of the output, we need a couples of contiguous rows form A as well as some columns form B. From now on, I will call "stripe" any sub-matrix that consist of a subset of contiguous rows or columns of its parent.

The good news is that the product of two strides can be parallelized: if we split the row stride in blocks and we do the same with the column stride, a wise distribution of blocks (uppermost and leftmost blocks assigned to the same node) will lead each processor to have a result block whose cells contains partial vector products, and so the final block can be obtained summing up all the local result blocks.

To implement this idea, each node will have a local matrix for C, as well as local A and B. These matrices represent the considered block of C we want to compute, an the stride parts required to obtain it. Again, this leads to the fact that non-master processors don't need the full A, B and C, so they can allocate them as 1×1 matrices and care just of their dimensions.

We are dividing the strides of A by columns, and the B strides by rows, so the key parameter we need to check to be divisible by the communicator size is `COL_N_A`. We also have restrictions over the other two parameters: they need to be divisible by the correspondent block dimensions. Practically, these three checks are required to be able to split our data as desired as desired, without have to deal with reminders.

Now, we are ready to analyze the core function: `matMulPar`. It is organized as a double `for` that iterates over the blocks of C, in the exact same manner I used for the parallel matrix transposition in homework 2. For each block, the operations to be performed are: distribute A, distribute B, recompose C. In practice, a node will use the same block of A for an entire row of C, so the distribution of A can be performed between the two iterations.

The data representation continue to play a major role. A needs to be split by rows, and so we can use a scatter, while B needs to be split by columns, and so we must send each line independently inside a `for`, just like the Row partition algorithm. Another complication is that a process can't send to itself, and so if the target process is the master, we have to manually copy the global B inside the local subB. Please note that I also played with `i_out` and `j_out` indexes to actually split just the analyzed stripes of A and B, and not the entire matrices.

Now, every node has its block of A and its block of B. If it multiplies them (with the classical algorithm) it will obtain its local C. Since we desire to sum these partial results, the natural choice is a `reduce`. A wise utilization of the indexes allows this reduction to be placed directly in the desired position of the matrix C memorized in the master.

1.3 Experiments and Benchmarks

1.3.1 The Automation Scripts

The scripts I used for this homework, follow exactly the same philosophy of the previous one. `compile.sh` creates the executables in a dedicated `bin` folder. The sequential program is compiled with `g++`, while the MPI programs are compiled with `mpicxx`. The `benchmark` script is in charge of launching the various binaries. For the serial code it's sufficient only one execution, while the parallel programs will be executed many times with a different number of processors. Just for readability reasons, I also made this script add an header to the report files that explain the meaning of the various columns included in it.

Another script is `clean.sh`, that removes the executables as well as the report files. It's purpose is to refresh your working environment before launching a new simulation. `run.pbs` is the script to manage the cluster. It loads the required modules, asks for the right amount and type of nodes, and print some information about them.

`plot_results` is a GNU octave script to convert the CSV report files into plots. it expects to find the `reports` folder as well as a `plots` folder. They are usually created by the benchmark script, but if for some case they don't appear you need to add them manually. So, the steps to reproduce my experiments are: first submit `rum.pbs` to the cluster, then transfer the reports into a machine with GNU Octave installed, and run `plot_results`. If you wish to retry the experiment, remember to launch `clean.sh`, otherwise the new measurements will be appended to the old files.

1.3.2 The Plots

For each case study, I created four plots: Strong Scaling, Weak Scaling, Speedup and Efficiency. The Strong Scaling represents over the x axis the number of nodes reported in logarithmic scale, and over the y axis the execution time of the algorithm launched with that number of processes. We expect this time to be smaller and smaller as the number of processors increases. I also reported as an horizontal line the time of the serial code, to have a baseline for comparison.

On the other hand, weak scaling is when ew do not just increase the size of our network, but we also enlarge the input dimension as well. This time we want the plot to be as similar as possible to an horizontal line. Again, I report the execution time of the serial code for comparison, with input dimension equal to the single processor case.

Speedup and Efficiency are concepts derivable from strong scaling. Speedup of an algorithm executed with a certain number of nodes is the ratio between its execution time and the time taken by the same algorithm when executed on a single processor. The ideal speedup is a diagonal line, that I reported also in the plot[3]. Efficiency is the next refinement of this concept. We start form speedup, and divide it by the number of processors used in the execution. This number is converted in percentage. The ideal efficiency is the 100% line.

1.3.3 The Chosen Parameters

In all the four programs I implemented, the size of the matrix has been the same: both A and B are 1024×1024, and so also C will be of this same dimension. This uniformity decision was taken to have a fair comparison between the different cases. For the block division, selected three representative sizes: 16×16, 64×64, 256×256. The logic behind this choice, is of course to try with very small blocks, blocks comparable with matrix size, and an intermediate dimension.

Another important parameter, is the number of processors. I spanned from one to $4 \times 64 = 256$, as this is the maximum number of processors I asked for in the PBS file. For simplicity, I run just codes whose amount of processors is a power of two. The master is set to be the rank 0, and the number of trials is set to two. This is because with 3 or higher values, the queue submission goes out of time for how I structured the benchmark.

[3]Remember that my plot is in logarithmic scale, and so the lined is deformed into an exponential

1.3.4 The Environment

I run the tests on the UniTN HPC cluster. In the `run.pbs` script I ask for 4 nodes with 64 processors each. Hence, I have 256 available MPI units. Of course, the more cores I have on a single device the better it is, because latency is a critical factor in distributed systems and if two processors are logically separated but physically in the same machine their communication will be much faster. In fact, for the first trials I asked for 16 nodes with 16 processors[4], and the result was much worse even if the total amount of processes is the same.

All the algorithms have been run in the same PBS submission, and hence in the same environment. This choice was again to enforce fairness of comparison. In the `output.o` file, I let the system print some information about such an environment with `lscpu` command. Just check it out if you are interested. In the same file, you will also find some information about the versions of the compilers `gcc`, `g++` and `mpicxx`.

1.4 Observations and Conclusions

1.4.1 Row Plot Analysis

I produced four groups plots: one for the row partition, one for the column partition, one to compare the three block partitions, and one to compare row case, column case and block case of size 256×256.

For the row partition (figure 1), we beautiful scalings up to 32 processors, and then a very rapid decline. The efficiency plot is the main testimony of this phenomena, as its is evident from its slope between 32 and 64. The strong and weak scaling confirm: the former starts to be horizontal after 32, while the latter has a sudden rise after that very same point.

This behavior can be explained with the communication overhead. When we have few nodes, the communication is much faster and so we have a negligible overhead that keeps the scaling close to the ideal. But when the number of processor increases, the communication overhead becomes bigger and bigger, until we reach a saturation point in which it becomes the main bottleneck of our system.

1.4.2 Column Plot Analysis

The column plot (figure 2), has the same shape, but it's even more accentuated: the efficiency plot stays beyond the 100% line up to 16 processors! of course, this is a theoretically impossible situation, that can only be explained by measurement fluctuations due to the small size of the considered matrices.

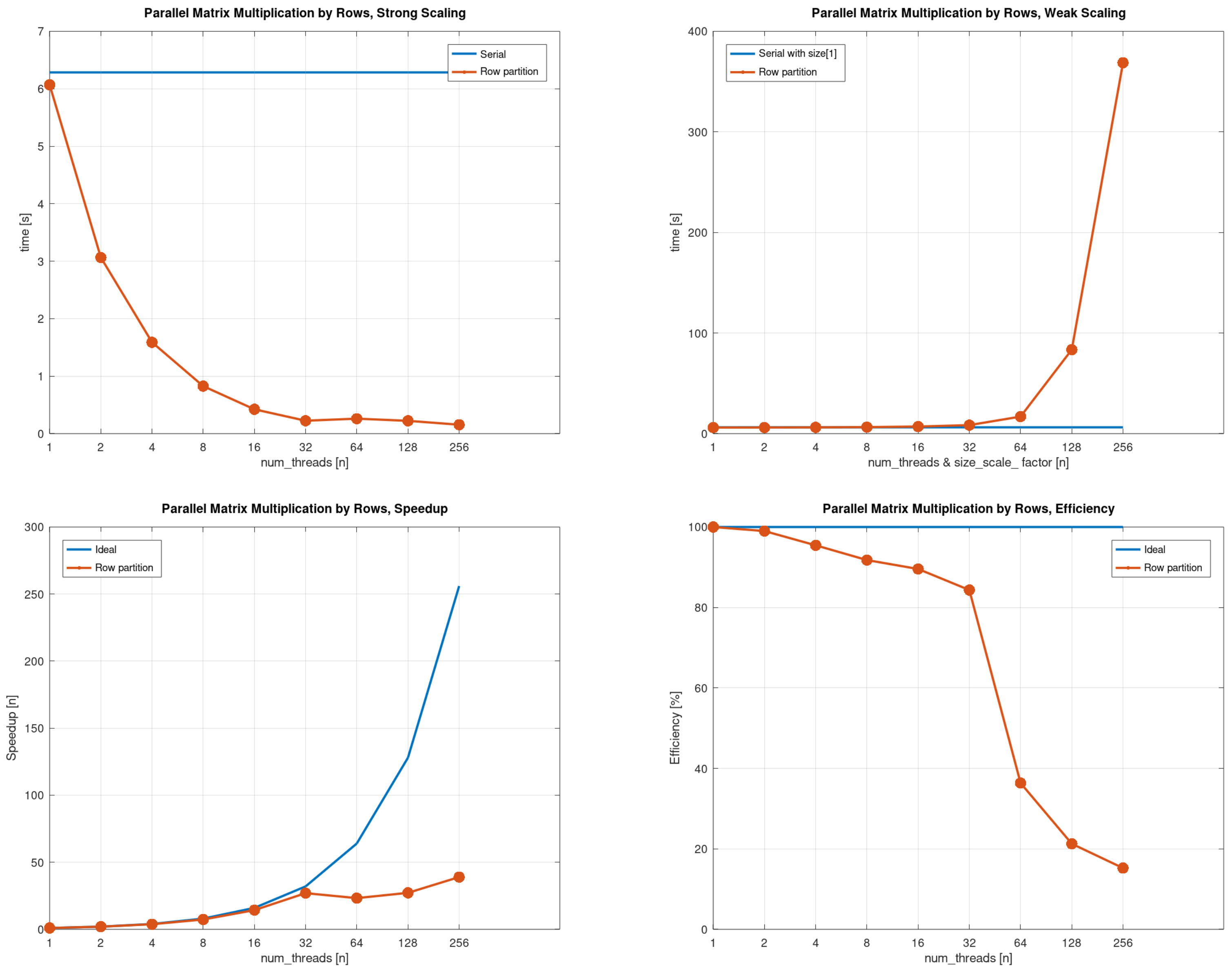
The weak scaling is again stable up to 32 MPI nodes, and then begin to rise. However, the strong scaling make us notice that for big sizes, the execution time doesn't just refuse to go down, but it increases. So, this data partition is very efficient for fe processors but even worse of the previous for big sizes.

1.4.3 Block Plot Analysis

The third plot (figure 3), allows us to compare the performance of different block sizes for the last algorithm. The first thing that comes to eye, is that bigger block means better performance. This is quite reasonable: If we have many small blocks, we will have more communications than a few

[4]I did this because in the UniTN HPC cluster we have few nodes with 64 cores, and so when you ask for them your request stay enqueued longer. I didn't want to waste time for that when I was just debugging.

Figure 1: Strong Scaling, Weak Scaling, Speedup and Efficiency of the data partition by rows.

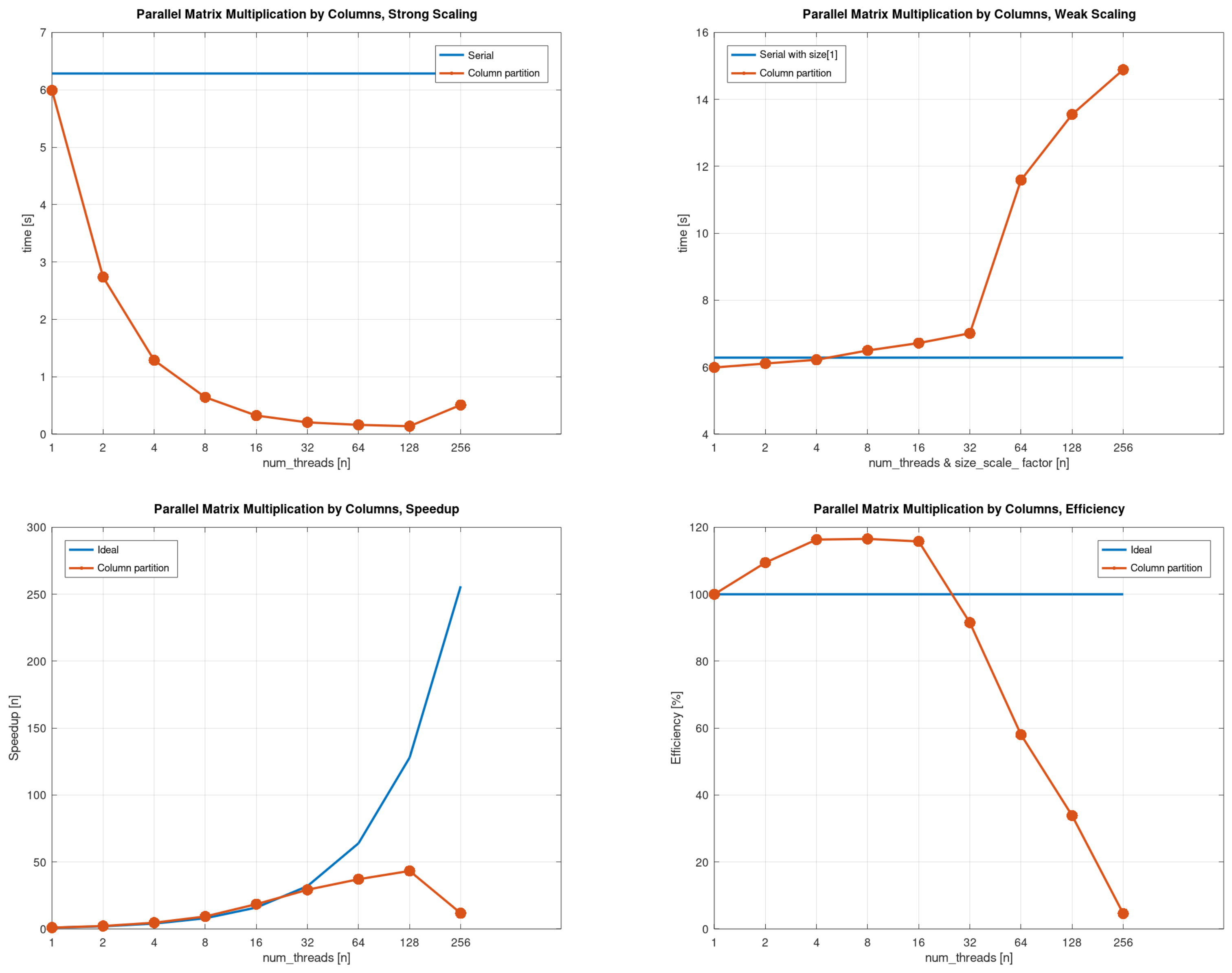


big ones and this will mean a bigger overhead. On the other hand, it's also reasonable to imagine tht when the block becomes toot big, the overhead to transmit the block will be the bottleneck and so performance will again start to decrease. In this plot we keep improving, and so me haven't reach that saturation point yet.

The strong scaling seems to confirm the 32 processor rule of thumb we have seen so far, while the weak scaling just tells us that the 16×16 block is so inefficient that its timing is orders of magnitude bigger than the other two cases. Also the strong plot confirm this assertion, as that block is the only one that does not just restart increasing its execution times after a point, but it also becomes slower than the serial algorithm.

The efficiency plot suffers again of the fluctuation problem, but apart from that it does not gives us particular hints about turning points in our domain, as it simply decreases almost linearly with no sudden slope change.

Figure 2: Strong Scaling, Weak Scaling, Speedup and Efficiency of the data partition by columns.

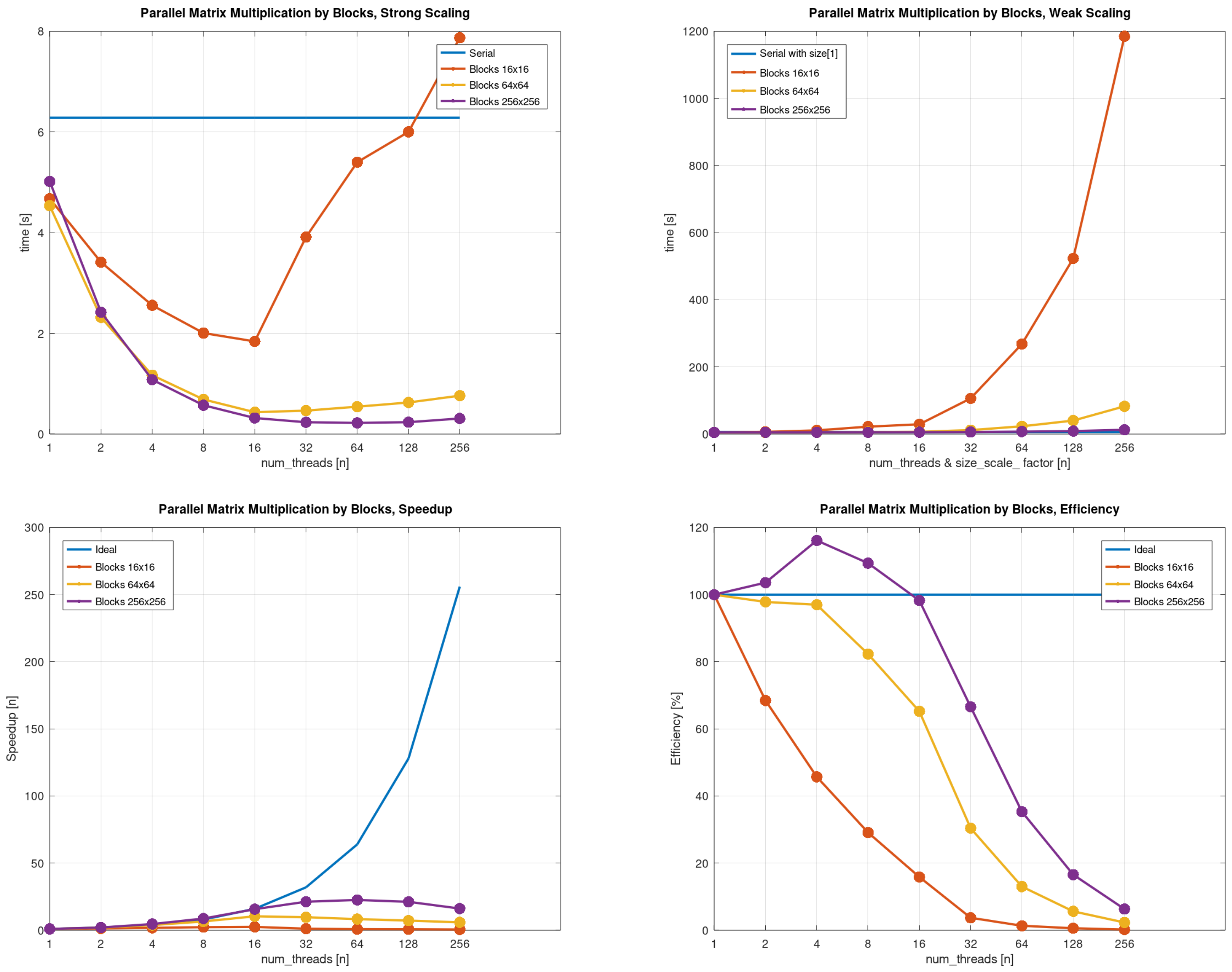


1.4.4 Comparison Plot Analysis

In the last plot (figure 4), I compare the performance of the three data partitions, choosing the bigger block as the representative of the block partition. Looking just at the strong scale and at the efficiency, we can see that the three approaches are almost equal to each other, and are all much better than the serial algorithm.

However, the weak scaling present an evident preponderance of the row partition, int he negative sense. This means that this data partition suffer from communication overhead much more than the other two. An explanation can be the following: in blocks and columns we are sending many small vectors (as C++ is row major) while in the row partition a few very long ones. So, we can conclude that unlike the block dimension analysis this time we have come up to a send dimension sufficiently high to let a few big transactions to cost more than many small ones.

Figure 3: Strong Scaling, Weak Scaling, Speedup and Efficiency of the data partition by blocks.

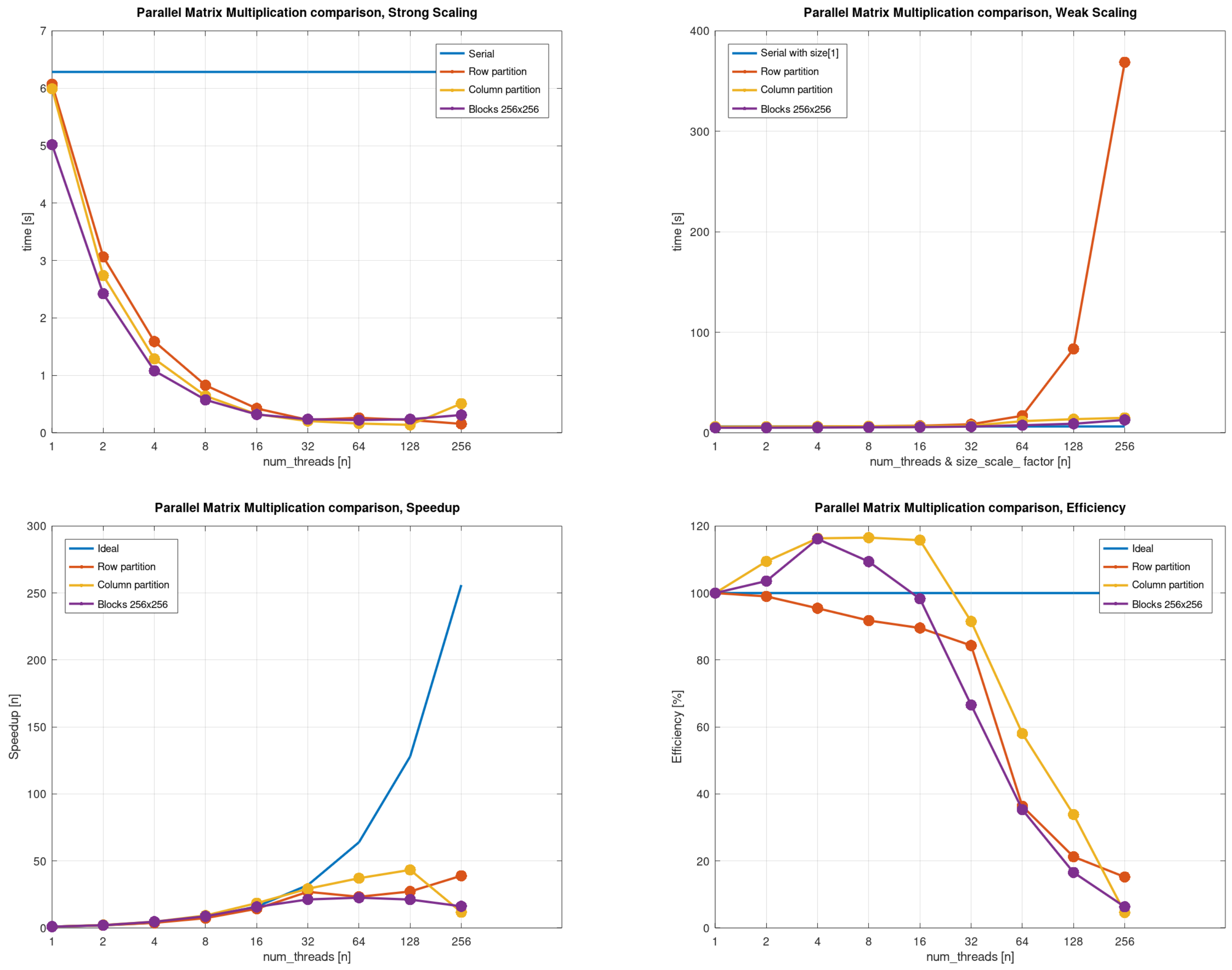


What about columns vs blocks? column algorithm is better than the block one for most of the points, however in the biggest size it starts to be worse. Furthermore, recall that in its analyses we noticed that it tent to rise again after a certain point, while the block case seemed to have this behavior less accentuate. So, my personal opinion is that for this small examples the column algorithm is still efficient, but for bigger problems the block algorithm will be more and more evidently the best.

1.4.5 FLOPS

With the Acronym FLOPS, we mean "Floating Point OPerationS", i.e. a number that indicates the quantity of operations between floating point variables our program needs to perform. Even taught the presented algorithms are quite different, behind the scene they all work with a triple for.

Figure 4: A comparison of the data partitions. It includes Strong Scaling, Weak Scaling, Speedup and Efficiency of the three algorithms, with 128×128 selected as the block dimension for the last partition.



So, the body of the innermost `for` needs to be multiplied by the limits dimensions of the iterations. The initialization of the value is just an assignment, not an operation, so we can ignore it. The real operations between floating point, are a sum and an multiplication. So, the number of flops is:

$$FLOPS = ROWN_A \times COLN_B \times COLN_A \times 2 = 1024 \times 1024 \times 1024 \times 2 = 2,147GFLOPS$$

This value can divided by the amount of time taken by each program to be executed, obtaining the FLOP/s, a measure to estimate how fast is the network in arithmetic tasks. I report here some values, the ones of the serial program and the ones of the parallel algorithms, in the strong scaling case with the maximum number of processes: 256.

$$serial : FLOP/s = 2,147 Giga \div 6.284910s \approx 341,6\, MFLOP/s$$

$$parallel_{rows} : FLOP/s = 2,147 Giga \div 0.155647s \approx 13,8\, GFLOP/s$$

$$parallel_{columns} : FLOP/s = 2,147 Giga \div 0.508608s \approx 4,2\, GFLOP/s$$

$$parallel_{blocks16x16} : FLOP/s = 2,147 Giga \div 7.872672s \approx 272,7\, MFLOP/s$$

$$parallel_{blocks64x64} : FLOP/s = 2,147 Giga \div 0.763254s \approx 2,8\, GFLOP/s$$

$$parallel_{blocks256x256} : FLOP/s = 2,147 Giga \div 0.309902s \approx 6,9\, GFLOP/s$$

1.4.6 Conclusions

For this exercise, I tried many different data partitions, and I learned that in the field of parallel computing it's difficult to know things a priori, but you have to dig into experiments. In fact, I initially had not idea of what to expect from the result, so I simply let them run and try to draw conclusions form them rather then use them to validate a theory I already had in mind. In my opinion, this is very coherent with the scientific method typical of experimental sciences like physics.

Furthermore, I learned that in parallel computing the result are very context dependent. Probably, the conclusion I got from this homework (i.e. that the block data partition is the best) will not be true for other problems, and so I will have to tailor a data partition more suitable for them. So, in summary I learned that in this field experiments play a major role.

2 Comparison of Parallelism Approaches

In this course, I have learned three parallelism approaches: implicit parallelism, OMP and MPI. I think that the best way to explain the relation between them, is “no pain no gain”. In fact, the experimental results confirm that there is no doubt that MPI is the most efficient solution, but the experience taught me that it is also the one that requires more attention and effort.

With implicit parallelization, the effort is as simple as write a flag. The performance gain is not so spectacular, but it is still much better than an unoptimized code. With OMP, the effort is still very low: a few lines of code, with few parameters to consider. The performance will be better than simple flags, but sometimes we risk to just add overhead and data races if not done carefully.

Finally, MPI have the best performance. The OMP and MPI experiment I run, have different matrix sizes and were probably on different nodes, however if we try to analyze the most context independent metrics i computed for both, i.e. the FLOP/s, We will se that OMP solutions were the order of 500 MFLOPS/s, while the solution implemented in this report with MPI overcomes the 10 GFLOPS/s. Unfortunatley, this power doesn't come for free: implement a MPI code require the entire reconstruction of your program. You can't do it as a fancy add on to implement if you have some time, but it must be your focus from the beginning to the end of your project.

So, in conclusion I fell that I will use the tools I learned in this course the following way: From now on, I will use optimization flags in every C++ program I will write, as they provide a non negligible optimization practically for free. I will think to OMP when I will be involved in a big project, and I will have some time left over. In that case, OMP will improve my efficiency, but I will not consume time that need to be delivered to thinking on the best algorithm.

Finally, I admit that I have the impression that MPI is used only in heavily parallelized codes, as you also need a network of computers that is not available, e.g., if I'm programming a desktop application. So, I think that I will use it in the future only when I will face programs related to cluster computing, such as scientific computation.

Finally, I feel to say that the moral of this course is that parallel computing algorithms and techniques are really, rally context dependent and there is no general recipe to solve them.

References

[1] For this homework, I found everything that I needed in the slides on the course.